LA-UR-12-24182
Approved for public release; distribution is unlimited.

Title: Validation of Two Hydrocodes with a Bi-Metallic Shaped Charge Experiment

Author(s): Ingraham, Daniel J.

Intended for: 2012 Computational Physics Summer Workshop Report

Validation of Two Hydrocodes with a Bi-Metallic Shaped Charge Experiment

Daniel Ingraham

August 16th, 2012

Abstract

Staggered grid (SGH) and cell-centered (CCH) Lagrangian Hydrodynamics are two approaches to modeling high explosives experiments. These experiments often involve complex flows with multiple materials with and without constitutive relationships. One example of complex flow phenomena is the discontinuous velocities along the tangential direction of a contact surface. Lagrangian methods combined with a contact surface algorithm are well-suited for these types of flows. In this work, the SGH and CCH schemes coupled with a contact surface algorithm are used to model a bi-metallic shaped charge experiment. The shaped charge experiment will be used to validate the contact surface methods, and is scheduled to be performed in the coming year at LANL. The experiment consists of a high explosive wrapped around a hemispherical shell of aluminum and an inner hemispherical shell of copper. The interface between the aluminum and copper shells is modeled both as a frictionless and "bonded" surface. The simulations are performed in a two-dimensional r-z coordinate system using the production code FLAG. A modified Gurney solution for an imploding sphere and Richardson extrapolation is used to evaluate the reasonableness of the results.

1 Introduction

1.1 What is a hydrocode?

Hydrodynamic codes (hydrocodes) are used to simulate complex phenomena involving multiple materials with complex equations of state and constitutive relationships experiencing large-scale deformation. The term "hydrocode" comes from the behavior of a material when it is subjected to a load many times greater than its strength — the material will flow much like a fluid, where the material's stress can be approximated by the "hydrodynamic" (pressure) component only. Most modern hydrocodes allow for constitutive relationships that model a material's strength, however. See refs. [1, 14] for some good introductions to hydrocodes.

Hydrocodes could be divided into two categories: Eulerian codes and Lagrangian codes. Eulerian codes solve the governing equations in a static frame of reference (or, in the case of moving grids, one that is defined ahead of time), with the material moving through the computational cells. Lagrangian codes, on the other hand, adopt a reference frame the *moves with* the deforming material. In an Eulerian code, the location of the mesh nodes and cells is constant, and thus the *volume* of each cell is constant. For a Lagrangian algorithm, however, it is the *mass* of each computational cell is constant, and the mesh node and cell locations are one of the quantities calculated by the algorithm. The mathematical difference between the two perspectives is essentially found in the form of the governing equations each code type solves: the Eulerian codes contain an advection term, while the Lagrangian codes do not. For example, the momentum equation in Eulerian form could be written as

$$\rho\frac{\partial v_i}{\partial t} + \rho v_j\frac{\partial v_i}{\partial x_j} = \rho f_i + \frac{\partial \sigma_{ji}}{\partial x_j}, \tag{1}$$

and the same equation in Lagrangian form as

$$\rho\frac{\mathrm{d}v_i}{\mathrm{d}t} = \rho f_i + \frac{\partial \sigma_{ji}}{\partial x_j}. \tag{2}$$

The term $\rho v_j \frac{\partial v_i}{\partial x_j}$ in (1) represents the net rate of momentum transfer per unit volume due to advection.

As with everything in life, both Eulerian and Lagranigan codes come with their own advantages and disadvantages. Most hydrodynamic calculations involve more than one material — thus at least one material interface is present. Because the mesh in a Lagrangian calculation moves with the material, the interface is sharply defined and handled in a natural way. In an Eulerian calculation the interface will become "blurred" as material of one type moves into cells originally containing only material of the "other" type.

One common difficulty with Lagrangian hydrocodes is "mesh tangling." As a Lagrangian mesh deforms, it may happen that the lines connecting the nodes may cross, which will prevent the calculation from continuing (see Figure 1). Eulerian codes do not have this problem, as the mesh remains stationary, or at least moves in a well-defined way that does not allow it to tangle. A related problem is one where the cells are compressed so greatly that the largest stable timestep allowed by the time-marching scheme falls below a "reasonable" limit (likely specified by the code user), again halting the simulation. (Hydrocodes generally use explicit time-marching algorithms, which generally have relatively restrictive stability limits.)

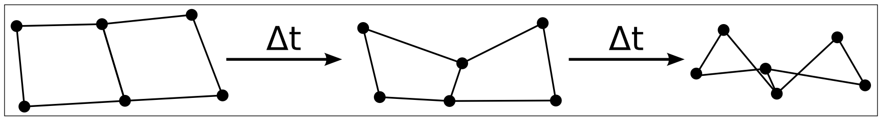


Figure 1: Example of a mesh becoming tangled during a Lagrangian calculation. The "Δt arrows" represent timesteps.

In this work, the LANL production hydrocode FLAG will be used. FLAG is actually an Arbitrary Lagrangian-Eulerian (ALE) code, meaning it can solve the flow equations in either (or both) reference frames. In this work, however, the Lagrangian solver of FLAG was used exclusively.

1.2 What is Solution Validation?

Solution validation is cleverly described by Bohem [3] and Blottner [2] (reported by Roache [11]) as ensuring one is "solving the right equations." The goal is to show that the equations a simulation code is solving is an adequate representation of reality (i.e. experiment). Performing validation studies assumes that the code is "solving the equations right" — Bohem and Blottner's definition of verification. Stated a slightly different way, one might say that the goal of validation is to estimate an quantity δ, where

$$\delta = u_{\text{nature}} - u_{\text{gov eqns}}. \tag{3}$$

Here, u_{nature} is the value of a quantity of interest of some system that actually exists in nature (the "real world"), uninfluenced by any error, and $u_{\text{gov eqns}}$ is the value of the same quantity of interest that would be found if the governing equations constructed to model the system were solved exactly. Equation 3 is inspired by Oberkampf and Trucano's discussion of validation [8]. δ can be broken up further:

$$\delta = (u_{\text{nature}} - u_{\text{experiment}}) + (u_{\text{experiment}} - u_{\text{gov eqns}}) \tag{4}$$

where $u_{\text{experiment}}$ is the value of the quantity of interest measured during an experiment. But (4) can be subdivided still further:

$$\delta = \underbrace{u_{\text{nature}} - u_{\text{experiment}}}_{E_1} + \underbrace{u_{\text{experiment}} - u_{\text{code}}}_{E_2} + \underbrace{u_{\text{code}} - u_{\text{gov eqns}}}_{E_3} \tag{5}$$

where u_{code} is the value of the quantity of interest found using the simulation code. Each of the underbraced terms is a component of error with a different interpretation. E_1 is the error in measurement of the quantity of interest during the experiment (the "experimental error"), E_2 is the difference between u found from experiment and the simulation code, and E_3 is the error created by discretizing the governing equations and solving them on a finite-precision computer (the "numerical error"). E_1 can be estimated with knowledge of the precision of the devices used to perform the experiment and measure the desired data, E_2 can be calculated directly from the experimental and simulation data, and E_3 can be found using Solution Verification techniques (e.g., Richardson Extrapolation or Roache's Grid Convergence Index). After these errors are quantified, an estimate of the errors arising from using the governing equation to model nature (i.e., δ) can be found with (5).

In reference to (5), the goal of this work is to find u_{code} and provide an estimate for $E_3 = u_{\text{code}} - u_{\text{gov eqns}}$ (in a general sense regarding u). It is hoped that the researchers performing the experiment will provide $u_{\text{experiment}}$ and $E_1 = u_{\text{nature}} - u_{\text{experiment}}$, and anyone with access to both the experimental and simulation data can find $E_2 = u_{\text{experiment}} - u_{\text{code}}$, and thus δ.

2 Numerical Methods

In this work, the LANL simulation code FLAG will be used for all simulations. FLAG is an arbitrary Lagranian-Eulerian (ALE) code, but only the Lagrangian solver was used in this work. Two different algorithms for solving the governing equations are implemented in FLAG and used in this work: the well-established compatible staggered grid hydrodynamics (SGH) scheme [5], and the new cell-centered hydrodynamics (CCH) scheme [4].

The SGH scheme, as the name implies, stores flow quantities on a staggered grid: position and velocity are located at the nodes, while density, stress, and internal energy are located at the cell centers. Because of this arrangement, the SGH method uses two control volumes in its spatial discretization of the flow equations: one for evaluating the momentum equation, and the other for evaluating an equation for the internal energy. The "semi-discrete" equations (i.e., discretized in space but not time) are shown in (6), and a diagram of the control volumes of the SGH scheme is show on the left side of Figure 2. Any suitable time marching scheme can be used to integrate the pseudo-ODEs in (6) — popular choices are the leapfrog or explicit second-order Runge-Kutta methods.

$$\begin{aligned}
\frac{\mathrm{d}M_c}{\mathrm{d}t} &= 0 \\
\frac{\mathrm{d}\vec{x}_p}{\mathrm{d}t} &= \vec{u}_p \\
M_p \frac{\mathrm{d}\vec{u}_p}{\mathrm{d}t} &= \sum_{i \in p} \sigma_c(i) \cdot d\vec{S}_i = \sum_{i \in p} \vec{F}_i \\
M_c \frac{\mathrm{d}\varepsilon_c}{\mathrm{d}t} &= -\sum_{i \in c} \vec{F}_i \cdot \vec{u}_{p(i)}
\end{aligned} \tag{6}$$

The CCH scheme, unlike SGH, stores the velocity of the material at the cell centers, along with density, stress, and total (not internal) energy. Thus the CCH scheme requires only one control volume to evaluate the "right-hand sides" of the semi-discrete equations found in (7).

$$\begin{aligned}
\frac{\mathrm{d}M_c}{\mathrm{d}t} &= 0 \\
\frac{\mathrm{d}\vec{x}_p}{\mathrm{d}t} &= \vec{u}_p^* \\
M_c \frac{\mathrm{d}\vec{u}_c}{\mathrm{d}t} &= \sum_{i \in c} \sigma_p^*(i) \cdot d\vec{S}_i \\
M_c \frac{\mathrm{d}j_c}{\mathrm{d}t} &= \sum_{i \in c} \sigma_{p(i)}^* \cdot d\vec{S}_i \cdot \vec{u}_{p(i)}^*
\end{aligned} \tag{7}$$

To sum the forces and work done on each cell's control surface, the CCH scheme uses a two-step process. First, the cell-centered velocity and stress values are projected to the nodes using a limited gradient. Next,

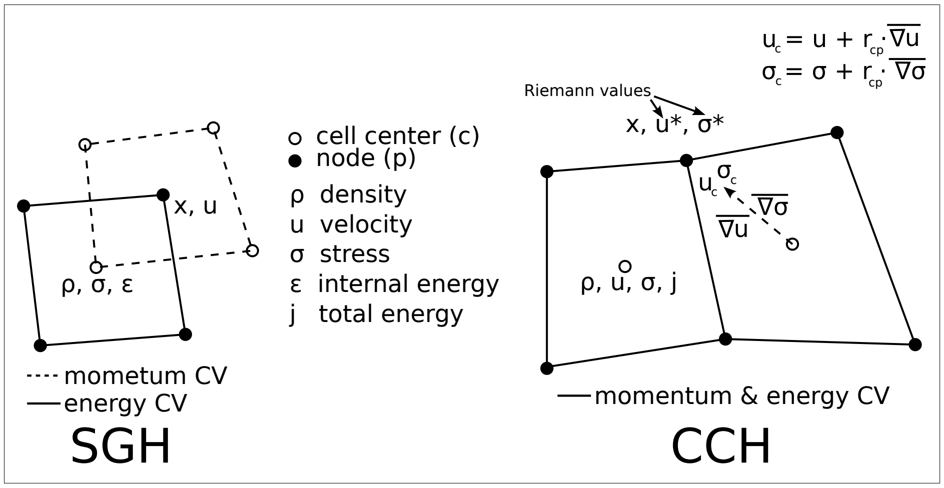


Figure 2: Control volumes and variable locations for the SGH (left) and CCH (right) schemes.

the projected values are used in a multi-dimensional Riemann-like solver to find the values of stress and velocity that will be summed around the cell control surfaces to find the temporal rate of change of velocity and total energy, i.e., the left-hand side of (7).

One difference between the SGH and CCH schemes is that SGH is *non-monotonic* — sharp flow discontinuities can lead to non-physical oscillations. This is not true of the CCH scheme. The practical effect of this difference will be seen in the results show later in this work.

Both the SGH and CCH schemes have contact surface algorithms that can be used to model the interface between two materials. Contact surface algorithms allow two materials to move relative to each other and separate if the materials are in tension. While the surfaces are in contact the components of stress and velocity normal to the interface are continuous, while those components parallel may be discontinuous. Figure 3 shows a conceptual diagram of a contact surface.

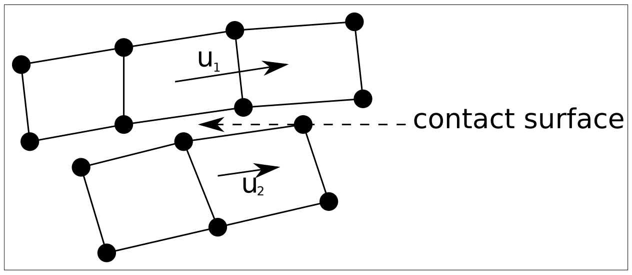


Figure 3: Diagram of a contact surface.

3 Shaped Charges

A shaped charge is a device used to penetrate a target using the energy stored in a high explosive (HE). Many different configurations of shaped charges exist, but they all share two common features: a mass of explosive material with a lined cavity at one end and a detonator at the opposite end. When the high explosive is detonated, the liner collapses, forming a dense, high-velocity “slug” of material that impacts and (hopefully) burrows deeply into the target. Figure 4 shows the nomenclature of a shaped charge. Refs. [13, 12] contain excellent introductions to shaped charges.

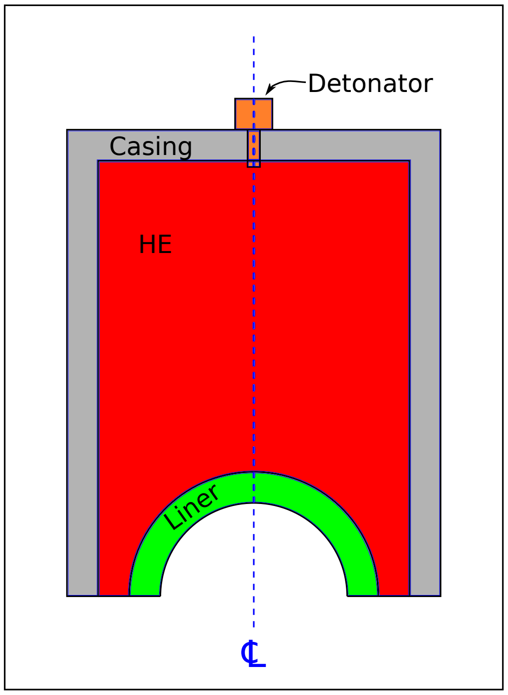


Figure 4: A diagram of a (cylindrical) shaped charge.

Shaped charges have numerous applications. They are used to “finish” oil wells, creating a channel that connects the well to the oil reservoir, and in demolition, especially for collapsing large structural columns. Perhaps shaped charges’ most well-known application, however, is their use in military munitions, such as missiles, torpedoes, and High Explosive Anti-Tank (HEAT) rounds.

4 Results and Discussion

4.1 Validation Case Setup

A diagram of the shaped charge geometry used in this work is shown in Figure 5. The geometry is defined in a r-z axis-symmetric coordinate system — z is an axis of symmetry, and thus the copper and aluminum liners are actually hemispherical shells. The extended 3mm-thick aluminum plate was added to prevent mesh tangling in the HE region; this allows the simulation to run longer. The air region below the shaped charge was added to avoid a “triple point”at the intersection of the bottom edge of the aluminum endplate, the contact surface line, and the air.

A justification for treating the copper-aluminum interface as a frictionless contact surface is warranted. If the following assumptions are made:

- Clearance between parts $\approx 1\mu m$
- Approximate viscosity of air $\approx 5.0 \cdot 10^{-5}\text{Pa} \cdot \text{s}$
- After-shock particle velocity $\approx 0.2\text{cm}/\mu\text{s}$

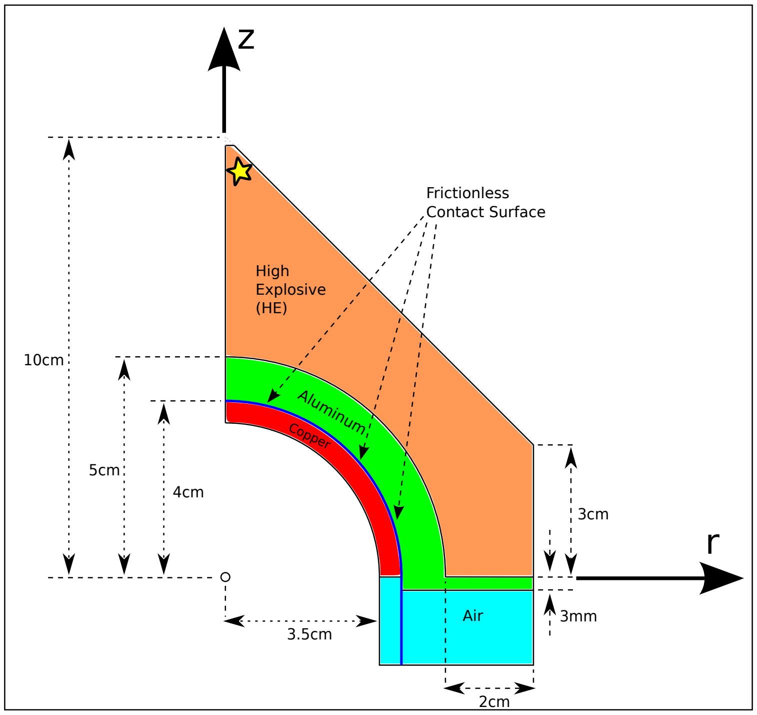


Figure 5: Diagram of the shaped charge geometry used in this work. The yellow star indicates the detonation point, and the blue curve is the locus of points where the contact surface algorithm is used.

then the shear stress at the copper-aluminum interface may be estimated as

$$\tau \approx \mu \frac{\Delta u}{\Delta y} = 5 \cdot 10^{-5}\text{Pa} \cdot \text{s} \frac{0.2 \frac{10^{-2}\text{m}}{10^{-6}\text{s}}}{1 \cdot 10{-}6\text{m}} = 10^5\text{Pa} = 1\text{bar}. \quad (8)$$

The shock pressure for the high explosive used here is about 0.25 MBars, which is five orders-of-magnitude larger than the estimation in (8). Thus the shock will completely overwhelm any frictional forces at the interface, and the assumption of a frictionless contact surface is reasonable. Runs with a "bonded" surface (i.e., completely attached) were performed and will be discussed, however.

A structured mesh was created using the mesh generation software Altair version `5.1.0`. Figure 6 shows a screenshot of the mesh. Ninety cells were used in the angular direction, and the radial spacing was set to $\frac{1}{12}$cm, giving approximately square cells in the "center" of the mesh (near the aluminum-HE interface). The mesh spacing in the r-direction was halved in the aluminum endplate and air region; this change was found to be more resistant to mesh tangling, allowing the simulation to run longer.

No casing was used in the shaped charge mesh. Some runs were performed using an outer aluminum casing, but it was found that the results differed little from runs without the casing, and actually made the simulation slightly less resistant to mesh tangling and cell collapse.

4.2 Results

The mesh of Figure 6 was used with the input files to simulate the detonation of the shaped charge. All results using the SGH scheme were run on LANL's "Yellow Rail" cluster and with FLAG version `3.2.Beta.2`; results using the CCH scheme were run on the "Turing" cluster and with FLAG version `3.3.Alpha.6`. Simulations were run in parallel, typically with the same number of processes as execution cores available on one cluster

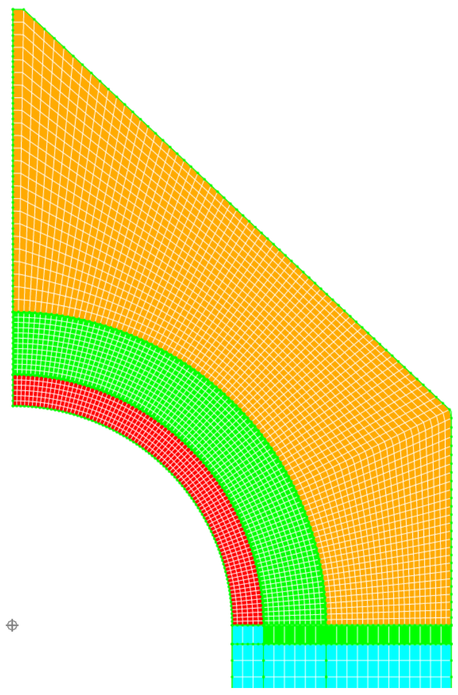


Figure 6: Screenshot of the shaped charge mesh created using Altair `5.1.0`.

node (i.e., eight for Yellow Rail and sixteen for Turing). All runs typically took significantly less than the five minutes of walltime requested from the job scheduler.

Figures 7 and 8 show the deformation of the shaped charge assembly for $t = 10\mu$s and $t = 20\mu$s, respectively. The results from the two schemes are qualitatively very similar.

Figure 9 shows a series of pressure maps for the SGH and CCH results. Detonation begins at the "top" (maximum z location) of the high explosive, causing a shock to propagate downward (Figure 9a). The high explosive region expands as it interacts with the free boundary condition. The shock wave impacts and transmits through the aluminum and copper liner, then eventually reaches the free boundary condition at the inner edge of the copper liner where it is reflected as an expansion wave (Figure 9b). The shock finally encounters the endplate and air, partially transmitting through to the air region and reflecting as another expansion wave (Figures 9c and 9d).

Some differences between the SGH and CCH schemes can be observed through Figure 9. Less of the shock appears to transmit through the liner in the CCH results. Perhaps because of this, the contact surface appears to open up wider with the SGH scheme. Two of the most striking differences between the two schemes, however, are seen in the copper material region. For the SGH scheme, "wiggles" are found in the copper liner trailing the shock. There was some question as to whether these oscillations were a numerical artifact of the SGH scheme, or a physical effect of the copper/aluminum interface opening and closing after the passing of the shock wave. To investigate this, the mesh with the same geometry but increased mesh spacing was created and run with the same inputs as the results from Figure 9. "Wiggles" were still observed, but with a larger wavelength. This result combined with the lack of the "wiggles" in the CCH results strongly indicates the oscillations are non-physical.

The other noticeable difference between the two schemes was also found in the copper liner, this time in the CCH results: pressure "spikes" were observed at the inner and outer edge of the copper liner. These are not seen in the SGH results, and the reader is reminded that the contact surface algorithm is active at the inner edge of the copper liner. Perhaps the interpolation of the stress and velocity with the limited gradient

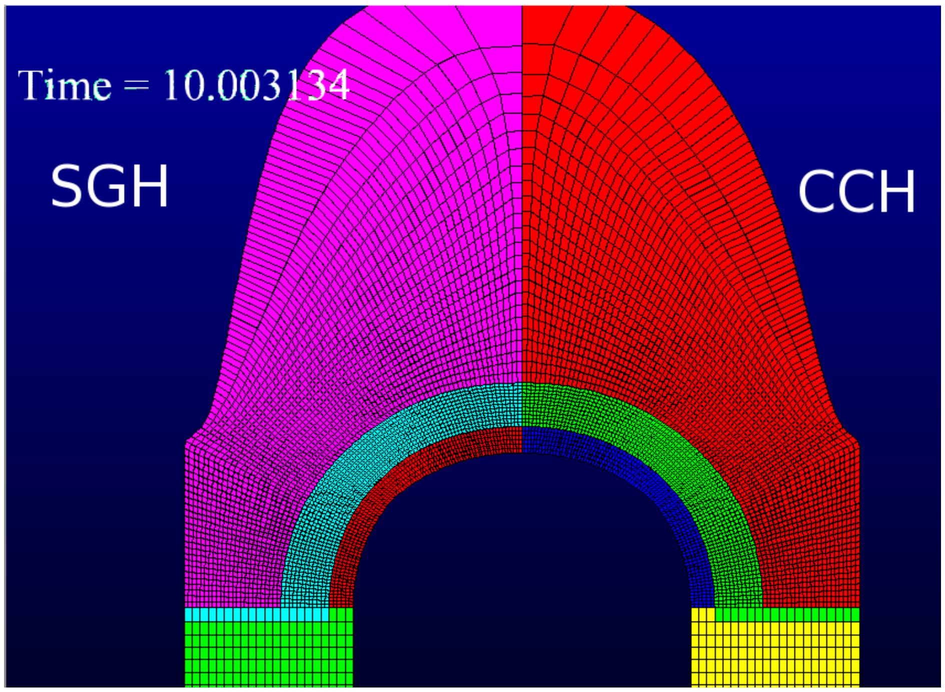


Figure 7: Comparison of the deformation of material at $t = 10\mu$s for the SGH and CCH schemes.

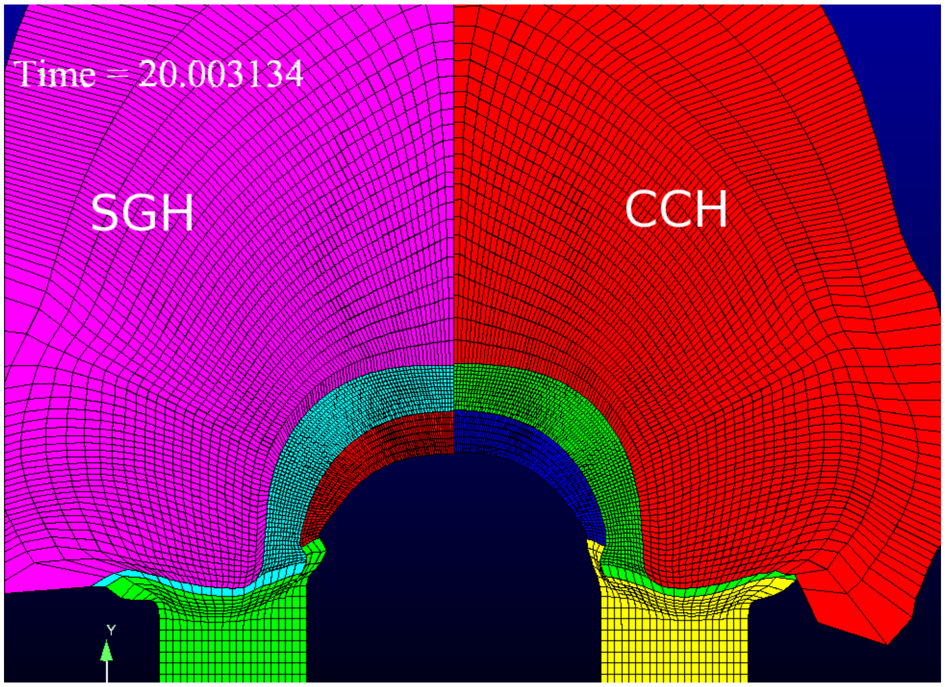
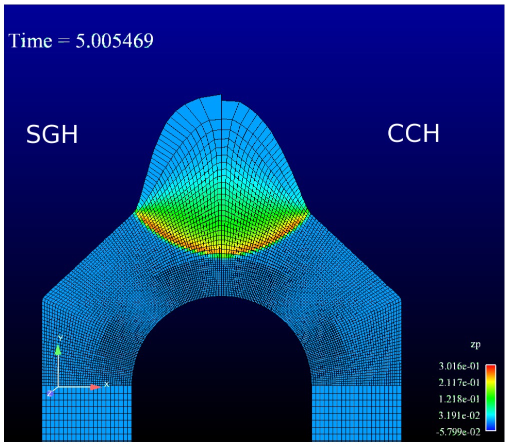
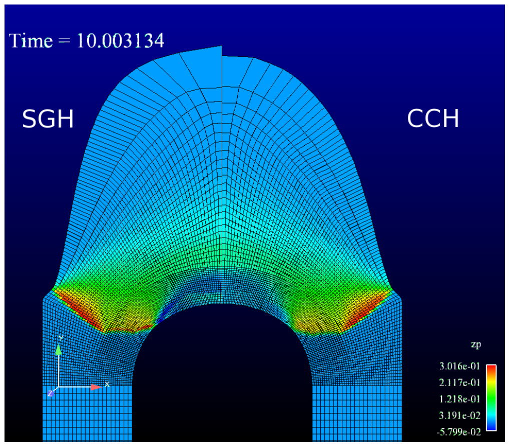


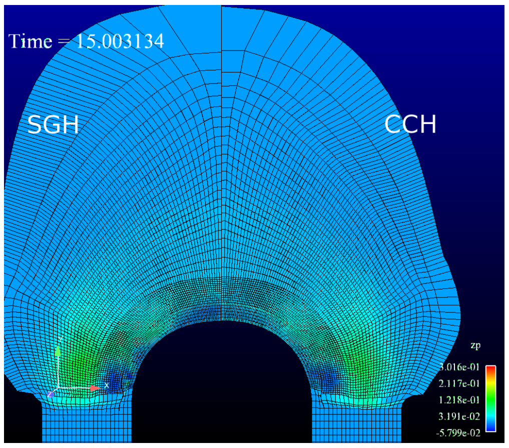
Figure 8: Comparison of the deformation of material at $t = 20\mu$s for the SGH and CCH schemes.



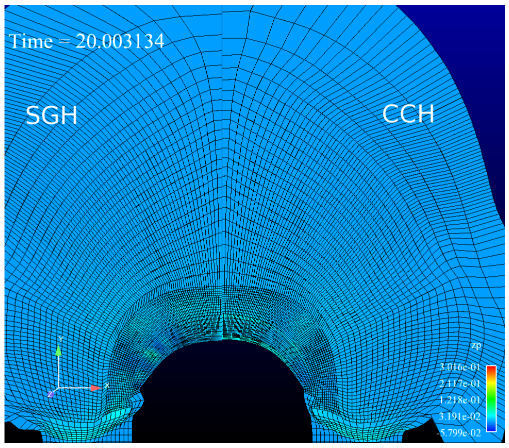
(a) $t = 5\mu$s



(b) $t = 10\mu$s



(c) $t = 15\mu$s

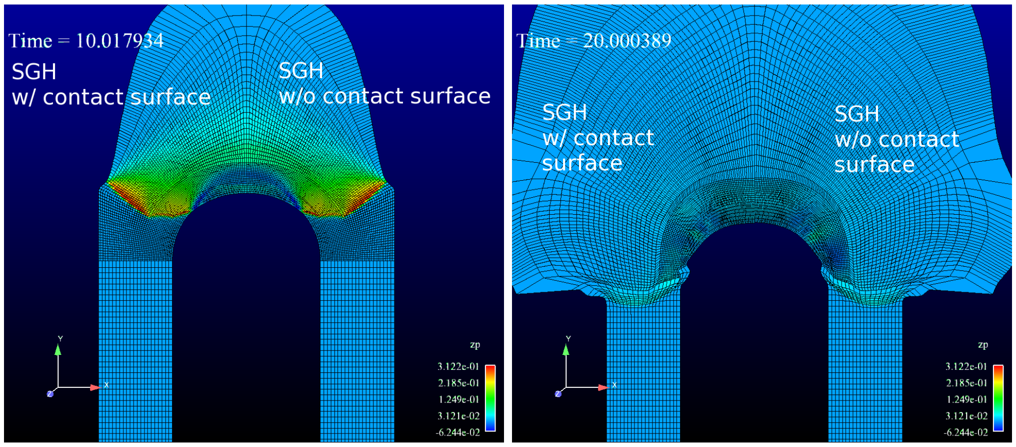


(d) $t = 20\mu$s

Figure 9: Sequence of pressure maps for the SGH and CCH runs. The units for the pressure scales are 10^2GPa, and time is shown in μs.

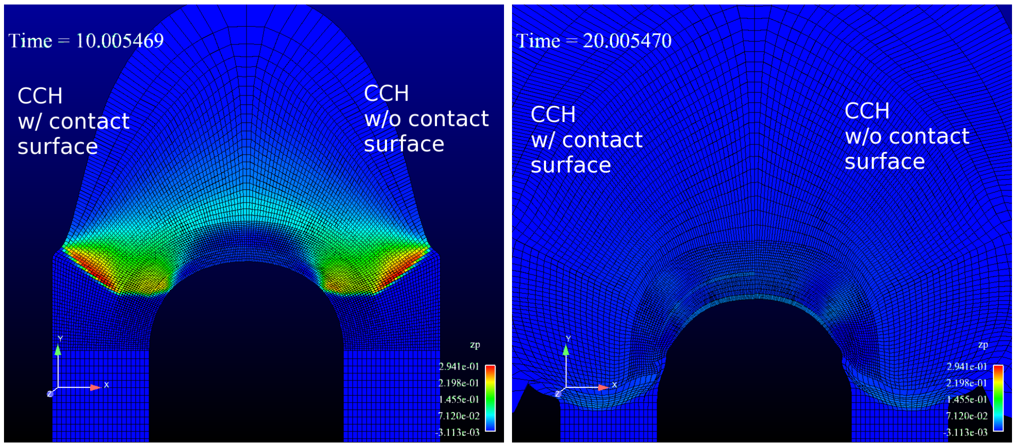
in the CCH scheme is responsible for these effects?

To investigate the effect of the contact surface, the same mesh and configuration from the results of Figures 7-9 was run without a contact surface at the copper/aluminum interface. A comparison demonstrating the effect of the frictionless contact surface is show for the SGH and CCH schemes in Figures 10 and 11, respectively. Overall the presence of the contact surface has little effect on the simulation results,



(a) $t = 10\mu s$ (b) $t = 20\mu s$

Figure 10: Effect of frictionless contact surface on the SGH runs.



(a) $t = 10\mu s$ (b) $t = 20\mu s$

Figure 11: Effect of frictionless contact surface on the CCH runs.

but some differences can be seen. Slightly more of the shock wave is transmitted through the "solid" copper/aluminum interface without the frictionless contact surface than with it. Perhaps because of this, the series of weak compression/expansions reflected after the detonation wave impacts the aluminum endplate and air is stronger for the case without the contact surface. The "wiggles" trailing the shock wave in the SGH results are about the same wavelength as the previous results, but slightly larger in amplitude (again likely due to the stronger shock in the liner region). The most significant observation from the comparison may be that the pressure spike at the copper/aluminum interface in the CCH runs has disappeared with the removal of the contact surface, strongly suggesting that the contact surface algorithm is responsible for this phenomenon.

4.3 Sanity Check: Modified Gurney Solution

Gurney solutions [6] use a control volume approach with a momentum and energy balance to estimate the "steady" velocity of a metal slab accelerated by the detonation of a high explosive. Gurney solutions are typically restricted to infinite or symmetric geometries and "explosive" configurations; however, Hirsch [7] extended the Gurney approach to implosive cylindrical and spherical cases. The spherical case will be used here to obtain a solution that can be compared with a slightly modified version of the simulation results presented in Section 4.2.

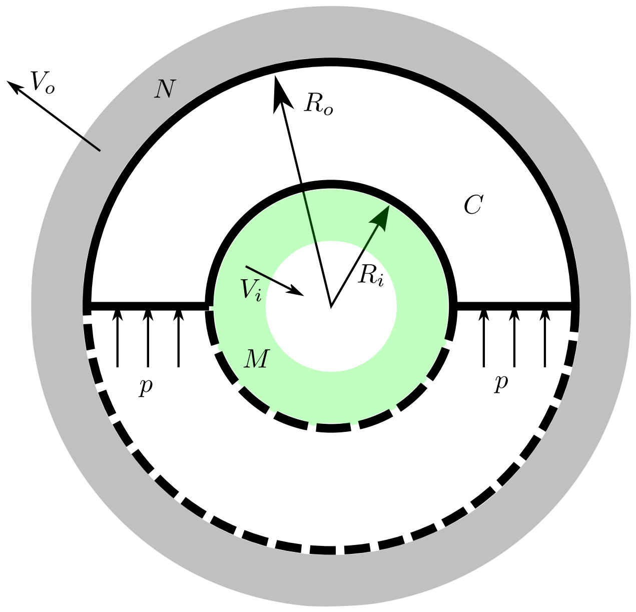


Figure 12: Diagram of Hirsch's modified Gurney solution for an imploding sphere. The bounding surfaces of the control volume are represented by solid lines, and the grey and green shaded areas denote the outer casing and inner liner. M, N, and C are the masses of the inner liner, outer casing, and explosive material, respectively.

Figure 12 shows the geometry and control volume associated with the Hirsch's imploding sphere solution. The momentum balance for the control volume is

$$\frac{1}{2}NV_o - \frac{1}{2}MV_i + 2\pi \int_{R_i}^{R_o} V_{\text{gas}}(r)\rho_{\text{ex}} R^2 \, dR = \int_0^\infty \int_{R_i}^{R_o} p(r,t)\, dr\, dt, \tag{9}$$

and the energy balance,

$$\frac{1}{2}CE = \frac{1}{2}\left(\frac{1}{2}M\right)V_i^2 + \frac{1}{2}\left(\frac{1}{2}N\right)V_o^2 + \pi \int_{R_i}^{R_o} V_{\text{gas}}^2 \rho_{\text{ex}} R^2 \, dR, \tag{10}$$

where ρ_{ex} is the density of the explosive and most of the other terms are defined in Figure 12. The E in (10) is the "Gurney energy" of the explosive material, which is defined as the amount of chemical energy (per unit mass) in the explosive that is converted into the kinetic energy of the liner, casing, and explosive products. As with all Gurney solutions, the velocity of the explosive product gasses V_{gas} is assumed to vary linearly with R:

$$V_{\text{gas}}(R) = (V_o + V_i)\frac{R - R_i}{R_o - R_i} - V_i. \tag{11}$$

Finally, the right-hand side of (9) (the impulse integral) is assumed to take the form

$$\int_0^\infty \int_{R_i}^{R_o} p(r,t)\,dr\,dt = a\pi m_o V_i \left(R_o^2 - R_i^2\right) \tag{12}$$

$$= a\pi m_o V_i R_i^2 \left(\left[\frac{R_o}{R_i}\right]^2 - 1\right) \tag{13}$$

$$= \frac{1}{2} a M \left(\left[\frac{R_o}{R_i}\right]^2 - 1\right) V_i \tag{14}$$

where m_o is the liner mass per unit projected area and a is a dimensionless constant of proportionality (taken to be unity here).

After considerable manipulation, equations (9)-(14) can be used to produce an expression for the liner velocity V_i:

$$V_i = \sqrt{2E} \Big/ \left[A\left\{\left(M/C + \frac{\beta^2+3\beta+6}{10(\beta^2+\beta+1)}\right)\Big/ A + A\left(N/C + \frac{6\beta^2+3\beta+1}{10(\beta^2+\beta+1)}\right) - \frac{3\beta^2+4\beta+3}{10(\beta^2+\beta+1)}\right\}\right]^{\frac{1}{2}} \tag{15}$$

where $\beta = R_o/R_i$ and $A = V_o/V_i$:

$$A = V_o/V_i = \left[M/C + a(M/C)(\beta^2-1) + \frac{\beta^2+2\beta+3}{4(\beta^2+\beta+1)}\right] \Big/ \left(N/C + \frac{3\beta^2+2\beta+1}{4(\beta^2+\beta+1)}\right). \tag{16}$$

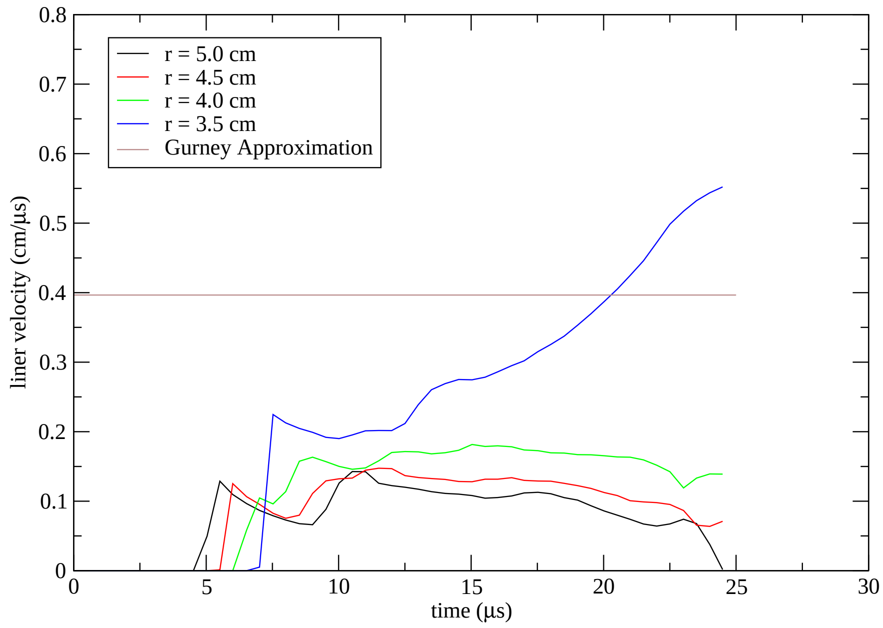


Figure 13: Velocity history of four evenly-spaced points in the all-aluminum liner of the modified shaped charge experiment compared to the Gurney solution for an imploding sphere. All points were 90° from the horizontal direction (i.e., at the "top" of the liner).

To use expressions (15) and (16) to predict the liner velocity V_i, the simulations from Section 4.2 had to be modified in two ways: the copper-aluminum liner was replaced by a all-aluminum liner, and the contact surface was removed. The mass and geometric parameters of the shaped charge geometry were substituted into (15) and (16) to obtain a value for V_i. The results are shown in Figure 13 for the SGH scheme. The liner velocity predicted by the Gurney solution was compared to the velocity of the aluminum in the modified simulation at four evenly-spaced points 90° from horizontal, i.e. the points furthest from the "edges" of the shaped charge, and thus hopefully far enough away from the sources of asymmetry to be a good candidate for comparison with the symmetric Gurney solution.

The result from the Gurney solution was at least of the same order of magnitude as the simulation results — a reasonably good comparison for such a "back of the envelope" approximation to the shaped charge configuration. There are numerous explanations for the discrepancy between the Gurney solution and simulation results for the liner velocity:

- The geometry of the shaped charge and imploding sphere are significantly different — the shaped charge liner is a hemisphere, not a sphere, and the shape of the high explosive is only vaguely spherical,
- The Gurney solution assumes a linear velocity distribution in the HE products — the simulation does not,
- The Gurney solution assumes the liner is instantly accelerated by the expanding HE gasses, and thus predicts a "steady-state" liner velocity, while the simulation obviously accounts for the unsteady motion of the shaped charge liner, and appears to not proceed long enough to predict the "steady-state" liner velocity (notice how the liner velocity for the $R = 3.5cm$ point is still trending upward at the end of the simulation).

Despite the above limitations, it is felt that the Gurney solution provides a useful "sanity check" for the simulations in the absence of experimental data.

4.4 Numerical Error Estimation: Richardson Extrapolation

As discussed in Section 1.2, it is the job of the researcher running the simulation code to provide some estimation of the numerical error present in the results of the simulation (i.e., E_3). Richardson extrapolation [9], as presented by Roache [10], can be used to provide such an estimation. One first begins with a Taylor series expansion of a solution at two different "discretization measures" h (i.e., mesh spacings or time steps):

$$f_\mathrm{f} = f_\mathrm{exact} + c_1 \left(h\right)^p + \mathrm{HOT} \tag{17}$$
$$f_\mathrm{c} = f_\mathrm{exact} + c_1 \left(rh\right)^p + \mathrm{HOT} \tag{18}$$

where f_f and f_c are the "fine" and "coarse" numerical solutions, r is the ratio of the coarse-to-fine discretization measures, p is the formal order-of-accuracy of the schemes used (here, $p = 2$), and HOT are higher-order terms. If one neglects HOT, then

$$f_\mathrm{c} - f_\mathrm{f} \approx c_1 (rh)^p - c_1 (h)^p \tag{19}$$
$$\approx c_1 h^p (r^p - 1) \tag{20}$$

and so

$$c_1 \approx \frac{f_\mathrm{c} - f_\mathrm{f}}{h^p \left(r^p - 1\right)} \tag{21}$$

and the leading error term for the coarse solution is

$$E_\mathrm{c} = c_1 \left(rh\right)^p = \frac{r^p \left(f_\mathrm{c} - f_\mathrm{f}\right)}{r^p - 1} \tag{22}$$

and for the fine,

$$E_\mathrm{f} = c_1 \left(h\right)^p = \frac{f_\mathrm{c} - f_\mathrm{f}}{r^p - 1}. \tag{23}$$

To use Richardson extrapolation, then, one must have (at least) two solutions for an identical simulation, differing only in discretization measure. For this work, the same shaped charge simulation described in Section 4.2 was used. The "fine" mesh was identical to that of Section 4.2, and the "coarse" mesh was like the fine, but with every other mesh line removed (i.e., $r = 2$). Any kind of solution could be taken as f — here the same data used in Section 4.3 was chosen at $t = 20.0\mu s$ (i.e., the liner velocity of four evenly-spaced points 90° from the horizontal). A constant Δt was used in the fine and coarse simulation runs to ensure that both simulations ended at the same time level.

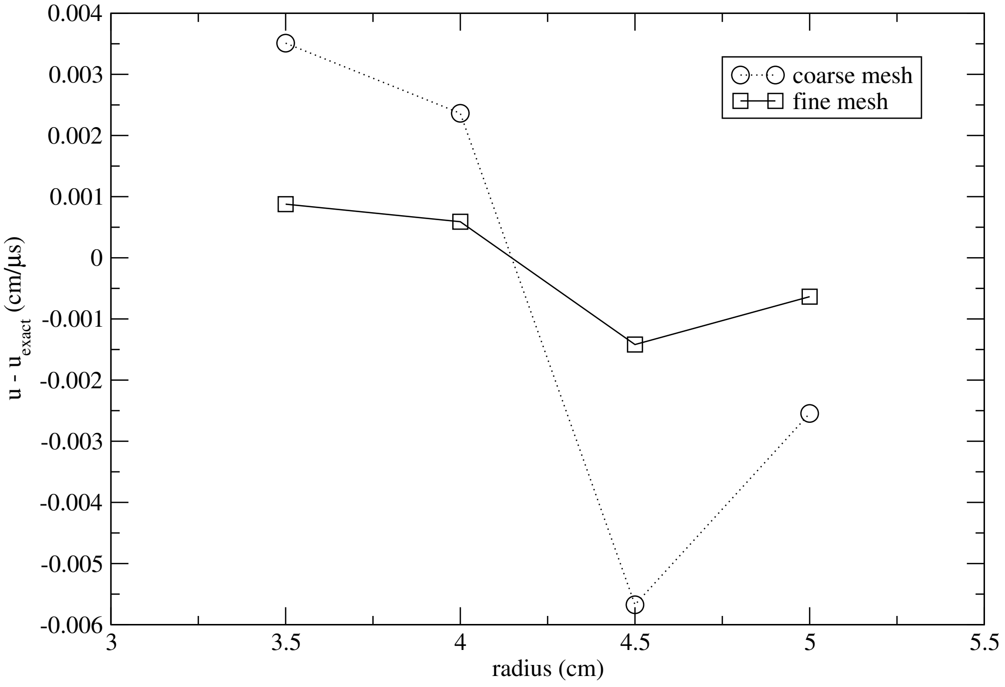


Figure 14: Liner velocity error as predicted by Richardson extrapolation.

Figure 14 shows the Richardson extrapolation results for SGH scheme. The magnitude of the error for the fine mesh is on the order of 10−3.

It must be noted that the use of Richardson extrapolation to estimate the error in a calculation implicitly assumes that the simulations are within the "asymptotic range," i.e., HOT from (18) are small compared to the leading error term. One method of determining this (brought to the attention of the author by Tyler Lung), would be to first run the same simulation for a range of h, and then plot f as a function of h^p, i.e.,

$$\begin{aligned} f(h^p) &= f_{\text{exact}} + c_1 h^p + \cancelto{0}{\text{HOT}} \\ f(y) &\approx f_{\text{exact}} + c_1 y. \end{aligned} \tag{24}$$

where $y = h^p$. If HOT are small, then the curve-fit of the f data plotted on a h^p scale will appear as a straight line, with the slope of the line being c_1 and the y-intercept an estimation of the exact solution. This linear region of the plot indicates the range of h for which the asymptotic range occurs.

5 Conclusions

In this work, the simulation results from a shaped charge validation test case were presented. After a brief explanation of the numerical methods used in the simulation code and the nature of a shaped charge device, the simulation results were summarized. The Staggered Grid Hydro (SGH) and Cell-Centered Hydro (CCH) schemes as implemented in LANL's FLAG code were were used to simulate the detonation of a bi-metallic shaped charge with and without a frictionless contact surface in r-z axisymmetric coordinates. Results between the two schemes were qualitatively similar, but some differences were observed. The SGH scheme produced "wiggles" in the solution trailing the detonation shock, and pressure "spikes" were found at the inner surface of the copper liner, and at the outer surface when the contact surface algorithm was used. A slightly modified shaped charge simulation was compared to a Gurney approximation of an imploding sphere. The Gurney approximation was within the same order-of-magnitude of the simulation data, indicating that the simulation results are not ridiculous. The numerical error of the simulation was estimated to be on the order of $10^{-}3$ using Richardson extrapolation.

Future work clearly includes comparing the simulation data to the forthcoming experiment at LANL. The pressure "spikes" at the inner and outer edge of the copper liner are (at least to the author) unexplained and worth investigating. More work could be done to quantify the numerical error and ensure that the simulations are within the asymptotic range as outlined at the end of Section 4.4. Finally, it may be worthwhile to explore using ALE to extend the life of the simulations — perhaps this would allow more opportunity to compare the simulations to the experimental data.

Acknowledgments

This work was performed during the 2012 Computational Physics Summer Workshop, XCP-4 at the Los Alamos National Laboratory, run by Dr. Scott Runnels. The author is thankful to Dr. Nathaniel Morgan (mentor, XCP-8) and Tyler Lung (workshop partner) for many useful discussions and frequent assistance.

References

[1] C.E. Anderson. An overview of the theory of hydrocodes. *International Journal of Impact Engineering*, 5(1):33–59, 1987.

[2] F.G. Blottner. Accurate navier-stokes results for the hypersonic flow over a spherical nosetip. *Journal of Spacecraft and Rockets*, 27:113, 1990.

[3] B.W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[4] D. Burton, T. Carney, N. Morgan, S. Runnels, S. Sambasivan, and M. Shashkov. A cell centered lagrangian godunov-like method for solid dynamics. *Submitted to the Journal of Computers and Fluids*, 2012.

[5] D.E. Burton. Consistent finite-volume discretization of hydrodynamic conservation laws for unstructured grids. In *Presented at the Nuclear Explosives Code Developers Conference (NECDC), Las Vegas, NV, 25-28 Oct. 1994*, volume 1, pages 25–28, 1994.

[6] R.W. Gurney. The initial velocities of fragments from bombs, shells and grenades. Technical Report BRL Report 405, Ballistic Research Laboratory, Aberdeen Proving Ground, Maryland, September 1943.

[7] E. Hirsch. Simplified and extended gurney formulas for imploding cylinders and spheres. *Propellants, Explosives, Pyrotechnics*, 11(1):6–9, 1986.

[8] W.L. Oberkampf and T.G. Trucano. Verification and validation in computational fluid dynamics. *Progress in Aerospace Sciences*, 38(3):209–272, 2002.

[9] L.F. Richardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A*, 210:307–357, 1911.

[10] P.J. Roache. Quantification of uncertainty in computational fluid dynamics. *Annual Review of Fluid Mechanics*, 29(1):123–160, 1997.

[11] P.J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, 1998.

[12] W. Walters and A.P. Ground. An overview of the shaped charge concept. In *11th Annual ARL/USMA Technical Symposium*, volume 5, 2003.

[13] W.P. Walters and J.A. Zukas. *Fundamentals of Shaped Charges*. John Wiley & Sons, 1989.

[14] J.A. Zukas. *Introduction to Hydrocodes*. Elsevier, 2004.

A Altair Script

This section contains the Python script used with Altair `5.1.0` to generate the shaped charge mesh shown in Figure 6.

```
import math
from altair import *

# For easier coding:
pi   = math.pi
tan  = math.tan
cos  = math.cos
sin  = math.sin
atan = math.atan

MeshName = "shaped_charge"
altair.startModel(model = MeshName)

# Number of cells in the theta direction.
NTheta = 90
dTheta = 0.5*pi/float(NTheta)

# Where everything begins.
origin = (0.0, 0.0)#cm

# Dimensions of the geometry.
RinnerCopper = 3.5#cm
RouterCopper = 4.#cm
RinnerAl = RouterCopper
RouterAl = 5.#cm
ChargeTotalHeight = 10.#cm
ChargeBottomWidth = 2.#cm
ChargeSideHeightGiven = 3.#cm
HalfTopFlatWidthGuess = 0.125#cm

# Number of cells in the radial direction in the copper part.
Nr_copper = int(round((RouterCopper - RinnerCopper) / (0.5 * (RouterAl + RinnerAl) * dTheta)))
print("Nr_copper = %s" % Nr_copper)

dr = (RouterCopper - RinnerCopper) / float(Nr_copper)

# Need to decide how many cells to have along the arc.
gammaGiven = tan(ChargeSideHeightGiven/(RouterAl + ChargeBottomWidth))
```

```
# Number of cells in the theta direction for the charge ``side flat.''
NThetaChargeSideHeight = int(round(gammaGiven / dTheta))
# So now I can find what the actual ChargeSideHeight will be.
ChargeSideHeight = (RouterAl + ChargeBottomWidth) * tan(NThetaChargeSideHeight*dTheta)

# Find the ``HalfTopFlatWidth''.
phiGuess = tan(HalfTopFlatWidthGuess / (ChargeTotalHeight - HalfTopFlatWidthGuess))
NThetaHalfTopFlatWidth = int(round(phiGuess/dTheta))
if NThetaHalfTopFlatWidth < 1:
    raise ValueError(
       "Can't create reasonable HalfTopFlatWidth with NTheta = %s. Try increasing NTheta." % NTheta
       )
HalfTopFlatWidth = (
    (ChargeTotalHeight * tan(NThetaHalfTopFlatWidth*dTheta)) /
    (1. + tan(NThetaHalfTopFlatWidth*dTheta))
    )

# The points for the ``flats'' of the copper shell (i.e., the two flat parts
# joining the inner and outer surface). I'm numbering them from left to right.
p1Copper = (RinnerCopper, 0.)
p2Copper = (RouterCopper, 0.)
p3Copper = (0., RinnerCopper)
p4Copper = (0., RouterCopper)

# The points for the ``flats'' of the aluminum shell (i.e., the two flat
# parts joining the inner and outer surface). I'm numbering them from left to
# right.
p1Al = (RinnerAl, 0.)
p2Al = (RouterAl, 0.)
p3Al = (0., RouterAl)
p4Al = (0., RinnerAl)

# There are lots of charge points (eight)! Here we go.
# p1Ch and p2Ch will form the ``flat'' for the ``imin'' surface of the charge
# block.
#       (r , z)
p1Ch = (RouterAl, 0.)
p2Ch = (RouterAl + ChargeBottomWidth, 0.)#cm
p3Ch = (RouterAl + ChargeBottomWidth, ChargeSideHeight)
p4Ch = (HalfTopFlatWidth, ChargeTotalHeight - HalfTopFlatWidth)
p5Ch = (0., ChargeTotalHeight - HalfTopFlatWidth)
p6Ch = (0., RouterAl)

# It might be best to use contour rather than line to create the outer edge of
# the charge. So that means I'll need a list of r and z locations.
outer_charge_rz = [p2Ch, p3Ch, p4Ch, p5Ch]

shell_spacing = 0.3 # 3mm

# Spacing in the ``horizontal'' (or r) direction for the air and aluminum
# endplate.
#dr_endplate = (RouterCopper - RinnerCopper) / 3. # This is the radial spacing
                                                  # for the previous runs.
# That was a bad idea. Changing it back.
dr_endplate = dr
# Maybe Dr. Morgan meant me to not refine the z direction? But that makes no
# sense... Hmm... I could *coarsen* the z direction:

# Spacing in the ``vertical'' (or z) direction for the air and aluminum
# endplate.
#dz = (RouterAl) * (0.5 * pi) / float(NTheta)
#dz = 1.25 * (RouterAl) * dTheta
dz = 3. * (RouterAl) * dTheta

# Theses are the points that I'll use to define the bottom aluminum shell.
p1AlShellBot_Cu = (p1Copper[0], -shell_spacing)
p2AlShellBot_Cu = (p2Copper[0], -shell_spacing)
p1AlShellBot_Al = (p1Al[0], -shell_spacing)
p2AlShellBot_Al = (p2Al[0], -shell_spacing)
p1AlShellBot_Ch = (p1Ch[0], -shell_spacing)
p2AlShellBot_Ch = (p2Ch[0], -shell_spacing)

p1AirBot_Cu = (p1Copper[0], -50*shell_spacing)
p2AirBot_Cu = (p2Copper[0], -50*shell_spacing)
p1AirBot_Al = (p1Al[0], -50*shell_spacing)
p2AirBot_Al = (p2Al[0], -50*shell_spacing)
p1AirBot_Ch = (p1Ch[0], -50*shell_spacing)
p2AirBot_Ch = (p2Ch[0], -50*shell_spacing)

# Initialize some lists that I'll be using.
inner_copper_arc_r = []
inner_copper_arc_z = []
outer_copper_arc_r = []
outer_copper_arc_z = []
outer_aluminum_arc_r = []
outer_aluminum_arc_z = []
#theta = []

N = 360 # number of points on the curves that we're going to make. This is not
# the same as the number of cells in the theta direction.
delta = 0.5 * pi / float(N - 1) # N - 1 because python starts at 0 and ends at len - 1
for i in range(N):
        inner_copper_arc_r.append(RinnerCopper * cos(float(i)*delta))
        inner_copper_arc_z.append(RinnerCopper * sin(float(i)*delta))

        outer_copper_arc_r.append(RouterCopper * cos(float(i)*delta))
        outer_copper_arc_z.append(RouterCopper * sin(float(i)*delta))

        outer_aluminum_arc_r.append(RouterAl * cos(float(i)*delta))
        outer_aluminum_arc_z.append(RouterAl * sin(float(i)*delta))

# Now make the contours.
inner_copper_contour = contour('InnerCopperSurface',
                               rList=inner_copper_arc_r,
                               zList=inner_copper_arc_z,
                               origin=origin)
outer_copper_contour = contour('OuterCopperSurface',
                               rList=outer_copper_arc_r,
                               zList=outer_copper_arc_z,
                               origin=origin)
inner_aluminum_contour = contour('InnerAluminumSurface',
                                 rList=outer_copper_arc_r,
                                 zList=outer_copper_arc_z,
                                 origin=origin)
outer_aluminum_contour = contour('OuterAluminumSurface',
                                 rList=outer_aluminum_arc_r,
                                 zList=outer_aluminum_arc_z,
                                 origin=origin)
inner_charge_contour   = contour('InnerChargeSurface',
                                 rList=outer_aluminum_arc_r,
                                 zList=outer_aluminum_arc_z,
                                 origin=origin)
outer_charge_contour = contour('OuterChargeSurface',
                                rzList=outer_charge_rz,
                                origin=origin)

# Next, the ``flats'' of the copper shell.
left_flat_copper  = line('LeftCopperFlat',  p1Copper, p2Copper)
right_flat_copper = line('RightCopperFlat', p3Copper, p4Copper)

# The ``flats'' of the aluminum shell.
left_flat_aluminum  = line('LeftAluminumFlat',  p1Al, p2Al)
right_flat_aluminum = line('RightAluminumFlat', p3Al, p4Al)

# This is the ``imin'' edge/surface of the charge.
left_flat_charge = line('LeftChargeFlat', p1Ch, p2Ch)
# This is the ``imax'' edge/surface of the charge.
right_flat_charge = line('RightChargeFlat', p6Ch, p5Ch)

# Adding a aluminum shell along the bottom of the shaped charge.
bottom_al_shell_Cu = line('BottomAluminumShell_Cu', p1AlShellBot_Cu, p2AlShellBot_Cu)
bottom_al_shell_Cu_rmin = line('BottomAluminumShell_Cu_rmin', p1Copper, p1AlShellBot_Cu)
bottom_al_shell_Cu_rmax = line('BottomAluminumShell_Cu_rmax', p2Copper, p2AlShellBot_Cu)

bottom_al_shell_Al = line('BottomAluminumShell_Al', p1AlShellBot_Al, p2AlShellBot_Al)
bottom_al_shell_Al_rmin = line('BottomAluminumShell_Al_rmin', p1Al, p1AlShellBot_Al)
bottom_al_shell_Al_rmax = line('BottomAluminumShell_Al_rmax', p2Al, p2AlShellBot_Al)

bottom_al_shell_Ch = line('BottomAluminumShell_Ch', p1AlShellBot_Ch, p2AlShellBot_Ch)
bottom_al_shell_Ch_rmin = line('BottomAluminumShell_Ch_rmin', p1Ch, p1AlShellBot_Ch)
bottom_al_shell_Ch_rmax = line('BottomAluminumShell_Ch_rmax', p2Ch, p2AlShellBot_Ch)

# Creating lines for air layer below aluminum shell.
bottom_air_Cu = line('BottomAir_Cu', p1AirBot_Cu, p2AirBot_Cu)
bottom_air_Cu_rmin = line('BottomAir_Cu_rmin', p1AlShellBot_Cu, p1AirBot_Cu)
bottom_air_Cu_rmax = line('BottomAir_Cu_rmax', p2AlShellBot_Cu, p2AirBot_Cu)

bottom_air_Al = line('BottomAir_Al', p1AirBot_Al, p2AirBot_Al)
bottom_air_Al_rmin = line('BottomAir_Al_rmin', p1AlShellBot_Al, p1AirBot_Al)
bottom_air_Al_rmax = line('BottomAir_Al_rmax', p2AlShellBot_Al, p2AirBot_Al)

bottom_air_Ch = line('BottomAir_Ch', p1AirBot_Ch, p2AirBot_Ch)
bottom_air_Ch_rmin = line('BottomAir_Ch_rmin', p1AlShellBot_Ch, p1AirBot_Ch)
bottom_air_Ch_rmax = line('BottomAir_Ch_rmax', p2AlShellBot_Ch, p2AirBot_Ch)

# Need to make some ``segments'', I guess.
inner_copper_segment = segment(inner_copper_contour)
outer_copper_segment = segment(outer_copper_contour)
```

```
inner_aluminum_segment = segment(inner_aluminum_contour)
outer_aluminum_segment = segment(outer_aluminum_contour)

inner_charge_segment = segment(inner_charge_contour)
outer_charge_segment = segment(outer_charge_contour)

left_flat_copper_segment  = segment(left_flat_copper)
right_flat_copper_segment = segment(right_flat_copper)

left_flat_aluminum_segment  = segment(left_flat_aluminum)
right_flat_aluminum_segment = segment(right_flat_aluminum)

left_flat_charge_segment  = segment(left_flat_charge)
right_flat_charge_segment = segment(right_flat_charge)

bottom_al_shell_Cu_segment = segment(bottom_al_shell_Cu)
bottom_al_shell_Cu_segment_rmin = segment(bottom_al_shell_Cu_rmin)
bottom_al_shell_Cu_segment_rmax = segment(bottom_al_shell_Cu_rmax)

bottom_al_shell_Al_segment = segment(bottom_al_shell_Al)
bottom_al_shell_Al_segment_rmin = segment(bottom_al_shell_Al_rmin)
bottom_al_shell_Al_segment_rmax = segment(bottom_al_shell_Al_rmax)

bottom_al_shell_Ch_segment = segment(bottom_al_shell_Ch)
bottom_al_shell_Ch_segment_rmin = segment(bottom_al_shell_Ch_rmin)
bottom_al_shell_Ch_segment_rmax = segment(bottom_al_shell_Ch_rmax)

bottom_air_Cu_segment = segment(bottom_air_Cu)
bottom_air_Cu_segment_rmin = segment(bottom_air_Cu_rmin)
bottom_air_Cu_segment_rmax = segment(bottom_air_Cu_rmax)

bottom_air_Al_segment = segment(bottom_air_Al)
bottom_air_Al_segment_rmin = segment(bottom_air_Al_rmin)
bottom_air_Al_segment_rmax = segment(bottom_air_Al_rmax)

bottom_air_Ch_segment = segment(bottom_air_Ch)
bottom_air_Ch_segment_rmin = segment(bottom_air_Ch_rmin)
bottom_air_Ch_segment_rmax = segment(bottom_air_Ch_rmax)

# Now, discretizations.

# This should discretize the ‘‘flats’’ evenly cells with ‘dr’ spacing.
left_flat_copper_segment.equalArcDistrib(dr)
right_flat_copper_segment.equalArcDistrib(dr)
left_flat_aluminum_segment.equalArcDistrib(dr)
right_flat_aluminum_segment.equalArcDistrib(dr)
left_flat_charge_segment.equalArcDistrib(dr)
right_flat_charge_segment.equalArcDistrib(dr)
bottom_al_shell_Cu_segment.equalArcDistrib(dr_endplate)
bottom_al_shell_Cu_segment_rmin.equalArcDistrib(dz)
bottom_al_shell_Cu_segment_rmax.equalArcDistrib(dz)
bottom_al_shell_Al_segment.equalArcDistrib(dr_endplate)
bottom_al_shell_Al_segment_rmin.equalArcDistrib(dz)
bottom_al_shell_Al_segment_rmax.equalArcDistrib(dz)
bottom_al_shell_Ch_segment.equalArcDistrib(dr_endplate)
```

```
bottom_al_shell_Ch_segment_rmin.equalArcDistrib(dz)
bottom_al_shell_Ch_segment_rmax.equalArcDistrib(dz)
bottom_air_Cu_segment.equalArcDistrib(dr_endplate)
bottom_air_Cu_segment_rmin.equalArcDistrib(dz)
bottom_air_Cu_segment_rmax.equalArcDistrib(dz)
bottom_air_Al_segment.equalArcDistrib(dr_endplate)
bottom_air_Al_segment_rmin.equalArcDistrib(dz)
bottom_air_Al_segment_rmax.equalArcDistrib(dz)
bottom_air_Ch_segment.equalArcDistrib(dr_endplate)
bottom_air_Ch_segment_rmin.equalArcDistrib(dz)
bottom_air_Ch_segment_rmax.equalArcDistrib(dz)

# This will divide the inner and outer surfaces of the shell into NTheta cells.
inner_copper_segment.equalAngleDistrib(NTheta)
outer_copper_segment.equalAngleDistrib(NTheta)
inner_aluminum_segment.equalAngleDistrib(NTheta)
outer_aluminum_segment.equalAngleDistrib(NTheta)
inner_charge_segment.equalAngleDistrib(NTheta)
outer_charge_segment.equalAngleDistrib(NTheta)

# Make some slides. Or not, for now.
outer_copper_segment.slide("slide1") # use this only.
#inner_aluminum_segment.slide("slide")
#outer_aluminum_segment.slide("slide2") # and this one.
#inner_charge_segment.slide("slide")

# Final step: make the blocks.
copper_block = block('CopperBlock',
                      iMin=left_flat_copper_segment,
                      iMax=right_flat_copper_segment,
                      iRes=len(inner_copper_segment),
                      jMin=inner_copper_segment,
                      jMax=outer_copper_segment,
                      #jRes=len(left_flat_copper_segment),
                      material='0003')

aluminum_block = block('AluminumBlock',
                      iMin=left_flat_aluminum_segment,
                      iMax=right_flat_aluminum_segment,
                      iRes=len(inner_aluminum_segment),
                      #jMin=inner_aluminum_segment,
                      jMin=outer_copper_segment,
                      jMax=outer_aluminum_segment,
                      #jRes=len(left_flat_aluminum_segment),
                      material='0002')

charge_block = block('ChargeBlock',
                      iMin=left_flat_charge_segment,
                      #iMax=right_flat_charge_segment,
                      #iRes=len(inner_charge_segment),
                      #feather='Distributed',
                      jMin=outer_aluminum_segment,
                      jMax=outer_charge_segment,
                      #jRes=len(left_flat_charge_segment),
                      material='0001')
```

```
aluminum_shell_Cu = block('AluminumShellCu',
                       #iMin=left_flat_copper_segment,
                       #iMax=bottom_al_shell_Cu_segment,
                       iMax=copper_block.iMin().coarsen(),
                       #iMin=bottom_al_shell_Cu_segment,
                       jMin=bottom_al_shell_Cu_segment_rmin,
                       jMax=bottom_al_shell_Cu_segment_rmax,
                       #iRes=1,
                       material='0004')

bottom_al_shell_Cu_segment_rmax.slide('slide1')

aluminum_shell_Al = block('AluminumShellAl',
                       iMax=left_flat_aluminum_segment.coarsen(),
                       iMin=bottom_al_shell_Al_segment.coarsen(),
                       jMin=bottom_al_shell_Cu_segment_rmax,
                       jMax=bottom_al_shell_Al_segment_rmax,
                       material='0002')

aluminum_shell_Ch = block('AluminumShellCh',
                       iMax=left_flat_charge_segment.coarsen(),
                       iMin=bottom_al_shell_Ch_segment.coarsen(),
                       jMin=bottom_al_shell_Al_segment_rmax,
                       jMax=bottom_al_shell_Ch_segment_rmax,
                       material='0002')

# Defining air blocks below the aluminum plate.
#air_Cu = block('AirCu',
                       #iMin=bottom_al_shell_Cu_segment,
                       #iMax=bottom_air_Cu_segment,
                       #jMin=bottom_air_Cu_segment_rmin,
                       #jMax=bottom_air_Cu_segment_rmax,
                       #material='0004')

air_Cu = block('AirCu',
                       iMax=aluminum_shell_Cu.iMin(),
                       #iMax=bottom_air_Cu_segment,
                       #dx1 = dr,
                       jMin=bottom_air_Cu_segment_rmin,
                       jMax=bottom_air_Cu_segment_rmax,
                       material='0004')

bottom_air_Cu_segment_rmax.slide('slide1')

air_Al = block('AirAl',
                       iMax=aluminum_shell_Al.iMin(),
                       #iMax=bottom_air_Al_segment,
                       #dx1 = dr,
                       jMin=bottom_air_Cu_segment_rmax,
                       jMax=bottom_air_Al_segment_rmax,
                       material='0004')

air_Ch = block('AirCh',
                       #iMin=aluminum_shell_Ch.iMin(),
                       iMax=aluminum_shell_Ch.iMin(),
                       #iMax=bottom_air_Ch_segment,
                        #dx1 = dr,
                        jMin=bottom_air_Al_segment_rmax,
                        jMax=bottom_air_Ch_segment_rmax,
                        material='0004')

# Finish up!
altair.endModel(dumpX3D=True,npes=30)

# vim:sw=4
```

B Gurney Solution Script

The Gurney solution described in Section 4.3 was evaluated using a Python script. The value of $\sqrt{E} =$ 2.90km/s was taken from the PBX-9404 entry in the table on page 51 of Ref. [13].

```
# This calculates the gurney solution for an imploding sphere.
from math import pi, sqrt

# Densities of the copper, aluminum and high explosive.
r_cu = 8.930 # g/cm3
r_al = 2.7 # g/cm3
r_he = 1.84 # g/cm3

# Volumes of the copper, aluminum, and high explosives
V_cu = 2.5 * pi # cm3
V_al = 6.0 * pi # cm3
#V_he = 59.0 / 3.0 * pi # cm3 # Error!
V_he = 178.0 * pi # cm3

# The Gurney velocity, sqrt(2*E), where E is the Gurney energy, or the amount of
# energy that is converted into kinetic energy of the liner and high explosive
# after detonation.
G = 2.9 # km/s

# b (beta in the reference I'm working off of) is the ratio of the outer and
# inner radius of the high explosive.
#b = (2.0 * sqrt(218.0)) / (5.0) # cm/cm
# Actually, beta might be the ratio of the outer and inner radius of the entire
# assembly. In that case, it would be
#b = (2.0 * sqrt(218.0)) / (3.5) # cm/cm
# From my better estimate of V_he, I calculated a ``effective hemispherical
# radius'' of the HE.
#radius_he_effective = (2. * 3./4. / pi * V_he)**(1./3.)
#print("radius_he_effective = %s cm" % radius_he_effective)
# Better ``effective hemispherical radius'' (earlier I was saying that the
# shaped charge was a *solid* hemisphere -- now I've removed the air and liner
# volume from the ``effective hemisphere'').
radius_he_effective = 7.87030458316
b = (radius_he_effective) / (3.5) # cm/cm

# a is a dimensionless constant of proportionality that is approximately unity in
# all the references I've seen.
#a = 1.0

# Mass of the liner. In the ``real'' shaped charge, there is an inner copper
# shell and outer aluminum shell, but here we'll say it's all aluminum.
M = r_al * (V_cu + V_al) # g

for a in [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
        1.9, 2.0, 2.5, 3.0, 3.5]:
#for a in [1.0,]:
    #for HE_vol_factor in [0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
            #1.9, 2.0, 3.0, 4.0, 5.0, 10.0, 20.0, 30.0]:
    for HE_vol_factor in [1.0,]:

        # Mass of the high explosive.
        C = r_he * V_he * HE_vol_factor # g

        #  A is an intermediate parameter that happens to be equal to the ratio of the
        #  outer and inner velocity of the high explosive.
        A =(
            (
                M/C
              + a*(M/C)*(b**2 - 1.0)
              + (b**2 + 2.0*b + 3.0)/(4.0*(b**2 + b + 1.0))
            )
            / ((3.0*b**2 + 2.0*b + 1.0)/(4.0*(b**2 + b + 1.0)))
           )

        # Now get what we're here for: the velocity of the liner.
        Vi =(
             G /
                sqrt(A * ((M/C + (b**2 + 3.0*b + 6.0)/(10.0*(b**2 + b + 1.0)))/A +
                    A*(6.0*b**2 + 3.0*b + 1.0)/(10.0*(b**2 + b + 1.0)) - (3.0*b**2 +
                        4.0*b + 3.0)/(10.0*(b**2 + b + 1.0))))) # same units as 'G'.

        # Convert the liner velocity from km/s to cm/microsecs.
        Vi = 0.1 * Vi
        print("HE_vol_factor = %s, a = %s, Vi = %s cm/microsecs" % (HE_vol_factor, a, Vi))
```